Threads

Name : Hamza Hassan Mohammed

Id : 26

Index

1- How the code is organized

- After reading the input files, we allocate memory for the A matrix and the B matrix and check if their sizes are valid for multiplication(Note that the two matrices are declared globally for easy access but their memory are allocated at runtime) .
- We allocate memory for the C matrix which will hold the result of the multiplication
- The program starts with trying to make a thread for each element in the result array and calculate the time for execution. Then It will make a thread for each row in the result array and calculate the time and the threads created.

2- Main Functions

- **Calculate one element**

```
void *calculate_one_element(void *data) {
    struct data *element_data = (struct data *) data;
    ll sum = 0;
    int row = element_data->row_number;
    int col = element_data->column_number;
    for (int i = 0; i < c1; i++) {
        sum += a[row][i] * b[i][col];
    }
    c[row][col] = sum;
    if (debug) fprintf(stderr, format: "c[%d][%d] = %lld \n", row, col, sum);
    pthread_exit( retval: NULL);
}
```

As the matrices are declared globally, we can access them easily from any function but the problem it that each thread has to know which element it is going to calculate; for that reason we define a struct to hold the row number and the column number.

- **Calculate one row**

```
void *calculate_one_row(void *data) {

    int *idx_ptr = ((int *) data);
    int i = (int) idx_ptr;


    for (int j = 0; j < c3; j++) {
        c[i][j] = 0;
        for (int k = 0; k < c1; k++) {
            c[i][j] += (a[i][k] * b[k][j]);
        }
    }
    pthread_exit( retval: NULL);

}
```

When we calculate the whole row in one thread, that thread has only to know the number of row it's going to calculate so we only pass an integer holding the value of the row.

3- Assumptions

- The files will be existed in the directory.
- The data inside the file are well formatted.

4- How to compile

there no complicated flags in order to compile the code. The only thing we need to add the ***pthread*** flag once we want to compile.

```
$ gcc main.c -pthread -o matmul.exe
```

5- Sample runs

```
hamza@Hamza-pc:~/CLionProjects/matmult$ gcc main.c -pthread -o matmul.exe
main.c: In function ‘calculate_one_row’:
main.c:34:13: warning: cast from pointer to integer of different size [-Wpointer-to-int-cast]
   34 |     int i = (int) idx_ptr;
      |             ^
main.c: In function ‘main’:
main.c:215:89: warning: cast to pointer from integer of different size [-Wint-to-pointer-cast]
  215 |         pthread_t row_thread = pthread_create(&row_threads[i], NULL, calculate_one_row, (void *) i);
      |                                                                                         ^
```

We will run the algorithm to calculate the multiplication of the following two matrices.

```
hamza@Hamza-pc:~/CLionProjects/matmult$ cat >in
row = 3 col = 3
1 2 3
4 5 6
7 8 9
^C
hamza@Hamza-pc:~/CLionProjects/matmult$ cat >in2
row = 3 col = 3
10 20 30
40 50 60
70 80 90
^C
```

```
hamza@Hamza-pc:~/CLionProjects/matmult$ ./a.out in in2 out
intput1 = in  intput2 = in2   outputfile = out
1 2 3
4 5 6
7 8 9
10 20 30
40 50 60
70 80 90
When using a thread for each entry ==> Data execution took 0 seconds and 4155 microseconds
Threads created = 9
When using a thread for each row ==> Data execution took 0 seconds and 378 microseconds
threads created = 3
```

We note that when using a thread for each element we have much more time than if we get a thread for each row.

```
hamza@Hamza-pc:~/CLionProjects/matmult$ cat out
The matrix using thread for each element
300 360 420
660 810 960
1020 1260 1500
The matrix using thread for each row
300 360 420
660 810 960
1020 1260 1500
```

The result of the matrices has been stored in 'out' file and eventually both solutions led to the same answer.

6- Comparison

At the sample run we notices that there is difference between both approached in terms of the time of execution, but both actually got the work done in less than one second.

We are going to multiply two matrices each of them have 100 row and 100 column to see if the difference is going to be more obvious.

```
hamza@Hamza-pc:~/CLionProjects/matmult$ ./a.out  bigin1 bigin2 bigout
intput1 = bigin1  intput2 = bigin2   outputfile = bigout
When using a thread for each entry ==> Data execution took 1 seconds and 47650 microseconds
Threads created = 10000
When using a thread for each row ==> Data execution took 0 seconds and 3176 microseconds
threads created = 100
```

Although the difference isn't noticeable, we can see that the first approach has executed in 1 second while the other one doesn't even exceed 4000 microseconds.